CS4212: mOOl Compiler - Team moang

Juan Manuel Munoz Perez A0134739X
Michael Noven A0146876M
Matthew Kennedy A0145811L

January 11, 2016

National University of Singapore

Abstract

After completing the frontend of the compiler, this final project aims to build the backend part which generates ARM code from the previous IR3. mOOL, a sub-language of Java, handled class hierarchy, methods overriding and overloading and methods/attributes modes access. However, implementing such characteristics might be very complex. This document focuses on the design of the mOOl language, a sub-language of mOOL which doesn't support the previous mentioned features, and gives an overview of how it is possible to implement the extra features. The final purpose is to run the mOOl compiler on an ARM processor.

Keywords: Compiler Design, OCaml

1 Building and running the mOOl compiler

In order to build the mOOl compiler, please apply the follow the steps:

1. Set up the ARM system by applying the indications in the project paper. We assume that global paths `NDK`, `SDK` and `CC` are set properly. In our example, it is:
   (a) `export NDK=.../android-ndk/r10e`
   (b) `export SDK=.../android-sdk/24.4.1`
2. Add to your path the SDK platform tools and newly created NDK toolchain created such that `$SDK"/platform-tools/":$NDK/tools/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin:$PATH`
3. Run the `make` shell command in order to build the `mOOL_compiler` executable file;
4. Launch the Android Virtual Device as stated in the assignment 3 paper;
5. Within the `mOOL_compiler` folder, run the shell script `mta.sh` (mOOl to ARM) like this: `./mta.sh -m mool_input_file` on OSX, or `sh mta.sh -m mool_input_file` on Windows using Cygwin. Such script generates the ARM machine instructions, compiles it using the gcc compiler and runs the bytecode in the Android virtual device.

2 mOOl design

2.1 OCaml pseudo-code algorithm for ARM code generation

The following section describes the design of the mOOl backend compiler. Receiving as an input the IR3 code (see `ir3mOOL_structs.ml`), the `ir3_to_ARM_non_opt.ml` file generates the ARM machine instructions. Below is the pseudo-code of the code generation:

```
M: list of md_decl3 objects
while M is not empty do:
   m = head(M)
   for each ir3_stmt3 in m do:
      ARM code generation from IR3
```

The full set of statements and expressions initially in the mOOL language are handled in the backend compiler. For instance, considering an object of class `A` with attributes `Int a1` and `Int a2` and method `ma` with signature `Int~Int`, the following mOOL line of code is allowed:

```
a.ma(a.a1, a.a2)
```

2.2 Dealing with offsets for variable access

The trickiest part of developing the backend compiler was correctly handling the offsets in order to access variables stored within the stack. This is done by keeping for each method a hash table with variables names as keys and values as their offset (see method `create_offset_tbl` in `ir3_to_ARM_non_opt.ml` file). To get the offset of a variable from the frame pointer, only a query on the hash table is needed.

2.3 Data/Machine instructions

In the `.s` files given as examples, we could notice the header `.data`. The previous category groups all contents accessed through a label. It is also mainly used for native type (string, integer, boolean) printing. Our algorithm generates a set of `data_instr` and a set of `machine_instr` which are then placed within the right tags. Our file code looks like this:

```
.data
data_instr
.text .global main
.type main, %function
machine_instr
```

2.4 The `new` object constructor

Assume an object `a` of class `A` has been declared. An initialisation statement with `new A()` must be done before `a`'s attributes and methods can be accessed. Indeed, `new` calls the external function `_Znwj` which allocates space within the heap for the current object a. So, accessing a's attributes is not possible without a prior initialisation.

3 Optimised version of mOOI

The previous backend compiler used a naive approach in order to allocate and assign registers to variables. The new optimised version uses the different techniques discussed in lecture. The optimisation flow is conducted as follows:

1. generate the basic blocks from the IR3 code
2. liveness analysis
3. optimise the basic blocks:
   (a) local optimisations
      i. dead code
      ii. redundancy
      iii. strength reduction
   (b) global optimisations
4. k-coloring for register alloc/assign
5. ARM code generation and optimisations:
   (a) peephole
   (b) tree-rewriting rules

3.1 Implementation of the flow graph

Below we give the data structure used in order to perform the optimisations on the IR3 code. For each method, we implemented a flow graph as a hash table consisting of (key, value) pairs:

- *key* would represent the label of the each block
- *value* is the basic block itself stored as an OCaml record with the following fields:
  * `id_babl`: identifier of the basic block;
  * `entry`: boolean indicating whether the current basic block is the entry point of the method;
  * `exit`: boolean indicating whether the current basic block is the exit point of the method;
  * `stmts`: hash table of the statements. The key is the statement number (row number) and the value is its IR3 code;
  * `succ`: list of successors basic blocks identifiers of the current basic block;
  * `pred`: list of predecessors basic blocks identifiers of the current basic block;
  * `live_in`: correspond to the IN_B set, i.e. the set variables alive while entring the current basic block;
  * `live_out`: correspond to the OUT_B set, i.e. the set variables alive while exiting the current basic block;
  * `def`: variables defined within the current basic block. This set is used to compute IN_B/OUT_B;
  * `use`: variables used within the current basic block. This set is used to compute IN_B/OUT_B;

Such data structure ease the access of the data in order to generate to optimise the IR3 code.

4 Features in mOOL that are not in mOOI

Class hierarchies are not supported by mOOI. This means a new class cannot inherit properties from one that already exists. If one would like to make use of methods and fields that are in other classes in a new class, then there are two options. Either re-write the methods and attributes in the current class, or define the other class with the desired properties in the same file. In most programming languages, this second option would not be available, but the lack of a hierarchy system in mOOI is mitigated by the loss of another common feature, access modes.

Access modes allow a programmer to restrict access to certain fields of an object. Since mOOI does not provide a way to force these limitations, encapsulation is not possible in its programs. Any class can make use of any method or field in the file, regardless of the specific class it was defined in.

Another consequence of the lack of hierarchies is that mOOI has no method overriding. When there are no parent classes there are neither any methods to inherit, nor any to override.

Unrelated to class hierarchies is the issue of method overloading. Methods with the same name, but different arguments are not allowed in mOOI.

5 Design of the missing features in mOOI

Below we explain how we would implement the extra features present in mOOL. We consider the code given in Appendix A (mOOL) and Appendix B (IR3 equivalent) to illustrate the discussion.

5.1 Class hierarchy

The static/type checker ouputs IR3 code of type `cdata3 list * md_decl3 * md_decl3 list`. Since the current lexer file doesn't handle the extends keyword, the lexer file from previous assignment can be used instead. Inheritance information will thus be contained within the class data and the inherited data will be fetched in the same way as the normal class attributes.

Inherited methods will also appear in the class data structure as shown in Appendix B, which means that the inherited method can be fetched from the method table. The mOOI compiler works with Static Dispatching, calling a method will refer to the declared type of the object before run time.

5.2 Method overriding/overloading

In IR3, each method is given an unique identifier which means that overriding methods can also be fetched from the method table by looking at the method signature. Overriding methods are already handled in the intermediate code generation, and are fetched as any other method. An example is shown in Appendix B, where method `(dummy~Bool, _DummyP_1)` is overriding the method in its super class.

5.3 Method/attribute modifiers

The access mode of all class attributes and method declarations have already been handled in the lexer and parser. Thus, private members would appear as any other attribute in the class data list and can be handled without any other change to the code generation file.

Appendix

A. mOOL code

```
class Main {

Void main() {
    Int t1;
    Int t2;
    Dummy a;
    DummyP d;

    a = new Dummy();
    d = a.getCompute();
    t1 = d.dummy(4); // 16
    return;
    }
}

class DummyP {
    Int i;

    Int square(Int k) {
        return k*k;
    }

    Int dummy(Bool b) {
        return 1;
}

class Dummy extends DummyP {
    Int j;

    Int dummy(Bool b2) {
        Bool b1;

        b1 = false;

        // return 1 if b2 is true
        if (b1 || b2) {
            return 1;
        }
        // return 0 if b2 is false
        else {
            return 0;
        }
    }

    DummyP getCompute() {
        DummyP a;

        a = new DummyP();
        return a;
    }
}
```

B. IR3 intermediate representation

```
======= Class3 =======

class3 Main{
parent:None;

----meth table----
  (main~Int~Int~Int~Int,main)}

class3 Dummy{
parent:DummyP;
  Int i;
  Int j;

----meth table----
  (square~Int,_DummyP_0);
  (dummy~Bool,_Dummy_0);
  (getCompute,_Dummy_1);
}

class3 DummyP{
parent:None;
  Int i;

----meth table----
  (square~Int,_DummyP_0);
  (dummy~Bool,_DummyP_1);
}

=======  CMtd3 =======

void main(Main this,Int i,Int a,Int b,Int d){
    ...
}

Int _Dummy_0(Dummy this, Bool b){
    ...
}

DummyP _Dummy_1(Dummy this){
    ...
}

Int _DummyP_0(DummyP this, Int k){
    ...
}

Int _DummyP_1(DummyP this, Bool b){
    ...
}
```